# pseudo Language Documentation

The language structure defines that each line must begin with a keyword token. There are three types of keyword token: active, inactive, and passive. Active keywords have active meaning, and they directly execute code. The active keywords are:

- increment
- decrement
- set
- input
- output
- call
- delay

Passive keywords cause other active keywords to be executed. The passive keywords are:

- if
- repeat
- function

Inactive keywords are simply used to denote separation between sets of arguments in active and passive keywords. The inactive keywords are shown below, along with their relevant active and passive keywords:

- set **to**
- repeat **times**
- input **to**
- output **to**

The other token types are string literal (denoted by ‘“‘ at each end of the string), integer literal (denoted by being made of only digit characters, boolean literal (denoted by ‘!’ at each end of the boolean), memory reference (integer enclosed in ‘[]’), variable reference (denoted by ‘_’ at each end of the name), and several partial tokens. The partial tokens are:

- ==
- <
- >
- !=

- NULL
- {
- }

The literal tokens and the memory and variable references are referred to as data tokens. Individual tokens are always separated by spaces. Only string literal tokens are allowed to have spaces within the token itself. Boolean literals are either ‘!true!’ or ‘!false!’.

The memory system is set up so that any memory reference can be initialised to a value. Memory locations are accessed via an integer index. If a memory location has already been initialised, only values of the same type can be assigned to it. The type can be changed by setting them to NULL before assigning the new value. Memory values act like tokens, they can take on the value of literal types.

Single line comments are supported; use ‘//‘ to denote the beginning of a comment.

The usage of each keyword is as follows.

```
// Will increase the integer value of the argument by one. Takes only one argument.

increment _integerVariable_; // Any pre-declared integer variable can be used
increment [0]; // Any memory reference can be used, unless it is already defined as being of a different type.


// Will decrease the integer value of the argument by one. Takes only one argument.

decrement _integerVariable_; // Any pre-declared integer variable can be used
decrement [0]; // Any memory reference can be used, unless it is already defined as being of a different type.


// Assigns a value to a variable or memory reference

set _variable_ to value; // Variable may or may not be initialised before the line. Value must be initialised, otherwise it must be a literal. If it is not a literal, it must be either a memory reference, a variable reference, or NULL.
set [0] to value; // The memory reference may or may not be
```

```
initialised before the line. Value must be initialised, otherwise
it must be a literal. If it is not a literal, it must be either a
memory reference, a variable reference, or NULL.


// Receives command line input as a string

input “Prompt here” to _variable_; // The type of the variable (if
already declared) must be string.
input “Prompt here” to [0]; // The type of the memory reference (if
already declared) must be string.


// Sends any value to the command line as a string, or to a
variable or memory location as a string

output “Value” [0] _var_ 201 !true! to _variable_; // The type of
the variable (if already declared) must be string. The number and
type of arguments is arbitrary, they are all concatenated as
strings. The result of this is then piped into the string value of
variable.
output “Value” [0] _var_ 201 !true!; // The number of arguments is
arbitrary, they are all concatenated. The result of this is then
piped into the command line.


// Executes the function with the same identifier as specified

call “function”; // The function must be already declared with the
correct identifier.


// Causes execution to wait for a number of milliseconds

delay 1500; // Currently can only be an integer literal time
specifier.


// Compares two values, and executes the code within the brackets
if the condition evaluates to true.

if value == value {…code…}; // Executes the code in the brackets if
end value of first value is equal to end value of second value.

if value > value {…code…}; // Executes the code in the brackets if
end value of first value is greater than end value of second value.

if value < value {…code…}; // Executes the code in the brackets if
end value of first value is less than end value of second value.
```

```
if value != value {…code…} // Executes the code in the brackets if end value of first value is not equal to end value of second value.


// Repeats the code inside the brackets a number of times

repeat value times {…code…} // Value can be an integer variable, an integer memory reference, or simply an integer literal


// Separates different segments of code and allows it to be executed more than once from different parts of the main program. Arguments can only be supplied through the use of other variables.

function “identifier” {…code…} // The function identifier must be specified as a string literal. The code inside will be skipped over when the function is declared, but the code will be returned to and executed when the ‘call’ keyword is used with the relevant identifier.
```